Algorithms for Monitoring Hyperproperties*

Christopher Hahn

Saarland University, Saarbrücken, Germany
hahn@react.uni-saarland.de

Abstract. Hyperproperties relate multiple computation traces to each other and thus pose a serious challenge to monitoring algorithms. Observational determinism, for example, is a hyperproperty which states that private data should not influence the observable behavior of a system. Standard trace monitoring techniques are not applicable to such properties. In this tutorial, we summarize recent algorithmic advances in monitoring hyperproperties from logical specifications. We classify current approaches into two classes: combinatorial approaches and constraint-based approaches. We summarize current optimization techniques for keeping the execution trace storage and algorithmic workload as low as possible and also report on experiments run on the combinatorial as well as the constraint-based monitoring algorithms.

Keywords: Hyperproperties · HyperLTL · Information-Flow · Monitoring · Runtime Verification.

1 Introduction

Hyperproperties [12] relate multiple computation traces to each other. Information-flow control is a prominent application area. Observational determinism, for example, is a hyperproperty which states that two executions agree on the observable output whenever they agree on the observable input, i.e., private data does not influence the observable behavior of the system. Standard trace monitoring techniques are not applicable to such properties: For example, a violation of observational determinism cannot be determined by analyzing executions in isolation, because each new execution must be compared to executions already seen so far. This results in a challenging problem: A naive monitor would store all traces and, thus, run inevitably out of memory. So how do we *efficiently* store, process and compare every executions seen so far? In this paper, we will give an overview on the significant algorithmic advances [1, 23, 24, 9, 31, 8, 7] that have been made in monitoring hyperproperties.

* This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (CRC 1223) and the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

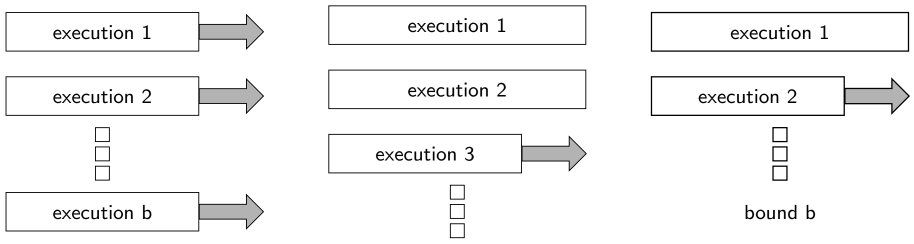

Fig. 1. Input Models: The parallel model (left), the unbounded sequential model (middle), and the bounded sequential model (right).

Monitoring hyperproperties requires, in general, extensions of trace property monitoring in three orthogonal dimensions: (1) how the *set* of execution traces is obtained and presented to the monitor, (2) how hyperproperties can be rigorously specified formally and (3) how algorithms process multiple traces at once without an explosion of the running time or storage consumption.

Input Model. There are three different straight-forward input models [25]: (1) The *parallel* model, where a *fixed* number of system executions is processed in parallel. (2) The *unbounded sequential* model, where an a-priori unbounded number of system executions are processed sequentially, and (3) The *bounded sequential model* where the traces are processed sequentially and the number of incoming executions is *bounded* (see Figure 1). Choosing a suitable input model for the system under consideration is crucial: The choice of the model has significant impact on the monitorability and, especially, on the monitoring algorithms. If, for example, the number of traces is a-priori bounded, offline monitoring becomes an efficient option. If, however, violations must be detected during runtime, algorithms must be specifically designed and optimized to reduce trace storage and algorithmic workload.

Hyper logical specifications. Hyperlogics are obtained by either (1) extending linear-time temporal and branching-time temporal logics with explicit trace quantification [11] or (2) by equipping first-order and second-order logics with the *equal-level predicate* [39, 29]. There are several extensions of logics for trace properties to hyperlogics (see [13] for a recently initiated study of the hierarchy of hyperlogics). HyperLTL [11] is the most studied hyperlogic, which extends linear-time temporal logic (LTL) with a trace quantification prefix. Let $\mathit{Out}, \mathit{In} \subseteq \mathit{AP}$ denote all observable output and input propositions respectively. For example, the HyperLTL formula

$$\forall \pi . \forall \pi' . \, \Big(\bigwedge_{o \in \mathit{Out}} o_\pi \leftrightarrow o_{\pi'} \Big) \, \mathcal{W} \Big(\bigvee_{i \in \mathit{In}} i_\pi \not\leftrightarrow i_{\pi'} \Big) \tag{1}$$

expresses observational determinism, i.e., that *all pairs* of traces must agree on the observable values at all times or until the inputs differ. With this added dimension, hyperlogics can relate traces or paths to each other, which makes

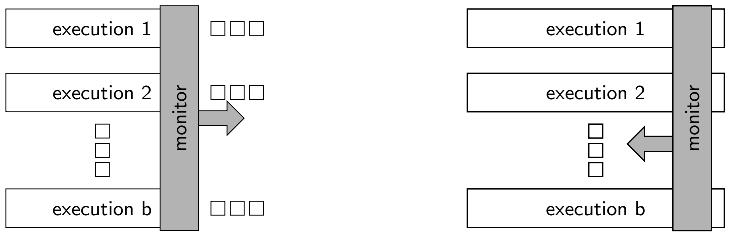

Fig. 2. [25] Monitor approaches for the parallel model: online in a forward fashion (left) and offline in a backwards fashion (right).

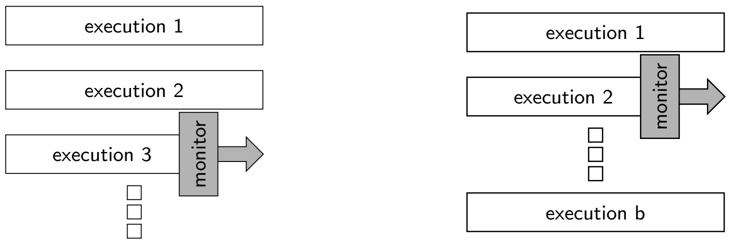

Fig. 3. [25] Monitor approaches for the sequential models: an unbounded number of traces (left) and bounded number of traces (right) are processed sequentially.

it possible to express hyperproperties, such as information-flow control policies rigorously and succinctly.

Algorithms. Current monitoring approaches can be classified into two classes: (1) algorithms that rely on combinatorial constructions, for example, on multiple instantiations of automaton constructions and (2) constraint-based algorithms that translate the monitoring requirements into Boolean constraints and, for example, apply rewriting techniques, which rely on SAT or SMT solving. Both types of monitoring techniques require heavy optimization, in order to make the monitoring problem of hyperproperties feasible. The bottleneck in combinatorial approaches is that a monitor needs to store, in the worst case, every observation seen so far. Optimizing the trace storage is therefore crucial. We describe a trace storage minimization algorithm that prunes redundant traces to circumvent this problem. Constraint-based approaches on the other hand, suffer from growing constraints, such that naive implementations push SAT and SMT solvers quickly to their limits. Keeping the constraint system as small as possible is therefore crucial. We report an optimization technique that stores formulas and their corresponding variables in a tree structure, such that conjunct splitting becomes possible. The algorithms reported in this paper in detail, i.e., [25, 31], have been implemented in the state-of-the-art monitoring tool for temporal hyperproperties, called RVHyper [24].

Structure. The remainder of this paper is structured as follows. We will report related work in Section 2 and give necessary preliminaries in Section 3. We classify current monitoring approaches into two classes in Section 4 and go exemplary into detail in [25] and [31]. We will summarize the optimization efforts that have been implemented in RVHyper in Section 5. In Section 6, we will report a summary of the experimental results that have been done over the last couple of years on RVHyper before concluding in Section 7.

2 Related Work

HyperLTL was introduced to model check security properties of reactive systems [11, 27, 26]. The satisfiability problem [19, 22, 20] and the realizability problem [21] of HyperLTL has been considered as well. For one of its predecessors, SecLTL [16], there has been a proposal for a white box monitoring approach [17] based on alternating automata. The problem of monitoring HyperLTL [6] was considered in an combinatorial approach in [1, 25] and in a constraint-based approach in [9, 31].

Runtime verification of HyperLTL formulas was first considered for (co-)k-safety hyperproperties [1]. In the same paper, the notion of monitorability for HyperLTL was introduced. The authors have also identified syntactic classes of HyperLTL formulas that are monitorable and they proposed a combinatorial monitoring algorithm based on a progression logic expressing trace interdependencies and the composition of an LTL_3 monitor.

Another combinatorial and automata-based approach for monitoring HyperLTL formulas was proposed in [23]. Given a HyperLTL specification, the algorithm starts by creating a deterministic monitor automaton. For every incoming trace it then checks that all combinations with the already seen traces are accepted by the automaton to minimize the number of stored traces, a language-inclusion-based algorithm is proposed, which allows for pruning traces with redundant information. Furthermore, a method to reduce the number of combination of traces which have to get checked by analyzing the specification for relations such as reflexivity, symmetry, and transitivity with a HyperLTL-SAT solver [19, 22], is proposed. The algorithm is implemented in the tool RVHyper [24], which was used to monitor information-flow policies and to detect spurious dependencies in hardware designs.

A first constraint-based approach for HyperLTL is outlined in [9]. The idea is to identify a set of propositions of interest and aggregate constraints such that inconsistencies in the constraints indicate a violation of the HyperLTL formula. While the paper describes the building blocks for such a monitoring approach with a number of examples, we have, unfortunately, not been successful in applying the algorithm to other hyperproperties of interest, such as observational determinism.

A sound constraint-based algorithm for HyperLTL formulas in the $\forall^2$ fragment is proposed in [31]. The basic idea is to rewrite incoming events and a given HyperLTL formula into a Boolean constraint system, which is unsatisfiable if a violation occurs. The constraint system is built incrementally: the algorithm starts by encoding constraints that represent the LTL constraints, which result from rewriting the event into the formula, and encode the remaining HyperLTL constraints as variables. Those variables will be defined incrementally when more events of the trace become available.

In [7], the authors study the complexity of monitoring hyperproperties. They show that the form and size of the input, as well as the formula have a significant impact on the feasibility of the monitoring process. They differentiate between several input forms and study their complexity: a set of linear traces, tree-shaped Kripke structures, and acyclic Kripke structures. For acyclic structures and alternation-free HyperLTL formulas, the problems complexity gets as low as NC.

In [8], the authors discuss examples where static analysis can be combined with runtime verification techniques to monitor HyperLTL formulas beyond the alternation-free fragment. They discuss the challenges in monitoring formulas beyond this fragment and lay the foundations towards a general method.

For certain information flow policies, like non-interference and some extensions, dynamic enforcement mechanisms have been proposed. Techniques for the enforcement of information flow policies include tracking dependencies at the hardware level [37], language-based monitors [36, 2, 3, 40, 5], and abstraction-based dependency tracking [30, 32, 10]. Secure multi-execution [15] is a technique that can enforce non-interference by executing a program multiple times in different security levels. To enforce non-interference, the inputs are replaced by default values whenever a program tries to read from a higher security level.

3 Preliminaries

Since hyperproperties relate multiple computation traces to each other, standard trace property specification logics like linear-time temporal logic (LTL) [34] cannot be used to express them. In this section, we will give a quick overview on how classic logics can be extended to obtain hyperlogics. We define HyperLTL, which is the, so far, most studied hyperlogic. We furthermore give the finite trace semantics of HyperLTL and define monitorability for the different input models.

3.1 Logics for Hyperproperties

Two extensions for obtaining hyperlogics are studied in the literature so far: (1) extending temporal trace logics, like LTL [34] and CTL* [18], with *explicit trace quantification* or (2) extending first-order and second-order logics with the *equal-level predicate* [39, 29]. An extensive expressiveness study of such hyperlogics has been initiated recently [13] and the hierarchy of linear-time hyperlogics is depicted in Fig. 4.

For example, HyperLTL extends LTL with trace quantification and trace variables. The formula

$$\forall \pi . \forall \pi' . \ \square \bigwedge_{a \in AP} a_\pi \leftrightarrow a_{\pi'} \tag{2}$$

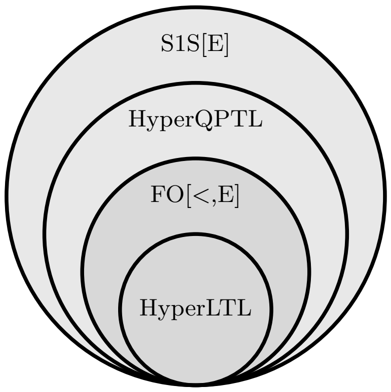

Fig. 4. The hierarchy of linear-time hyperlogics [13].

expresses that *all pairs* of traces must agree on the values of the atomic propositions (given as a set AP) at all times.

The other technique for obtaining hyperlogics consists of adding the equal-level predicate E, which relates the same time points on different traces. The HyperLTL formula (2), for example, is equivalent to the FO$[<, E]$ formula

$$\forall x. \forall y.\ E(x, y) \rightarrow \bigwedge_{a \in AP} (P_a(x) \leftrightarrow P_a(y)).$$

Solving the runtime verification problem for logics beyond HyperLTL is still open. Current monitoring approaches focus on the, so far, best understood temporal hyperlogic HyperLTL, which we will define in the following.

3.2 HyperLTL

Let AP be a set of *atomic propositions.* A *trace* t is an infinite sequence over subsets of the atomic propositions. We define the set of traces $TR := (2^{AP})^\omega$. A subset $T \subseteq TR$ is called a *trace property.* A *hyperproperty* H is a set of trace properties, i.e., $H \subseteq \mathcal{P}(\Sigma^\omega)$. We use the following notation to manipulate traces: let $t \in TR$ be a trace and $i \in \mathbb{N}$ be a natural number. $t[i]$ denotes the i-th element of t. Therefore, $t[0]$ represents the starting element of the trace. Let $j \in \mathbb{N}$ and $j \geq i$. $t[i, j]$ denotes the sequence $t[i]\ t[i + 1] \ldots t[j - 1]\ t[j]$. $t[i, \infty]$ denotes the infinite suffix of t starting at position i. Let $\mathcal{V}$ be an infinite supply of trace variables.

The syntax of HyperLTL is given by the following grammar:

$$\begin{aligned} \varphi &::= \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi \ \text{, and} \\ \psi &::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi\, \mathcal{U}\, \psi \ , \end{aligned}$$

where $a \in AP$ is an atomic proposition and $\pi \in \mathcal{V}$ is a trace variable. The quantification over traces makes it possible to express properties like "on all traces ψ must hold", which is expressed by $\forall\pi.\ \psi$ and, dually, that "there exists a trace such that ψ holds", which is denoted by $\exists\pi.\ \psi$. The derived operators $\Diamond$, $\Box$, and $\mathcal{W}$ are defined as for LTL.

A HyperLTL formula defines a *hyperproperty*, i.e., a set of sets of traces. A set T of traces satisfies the hyperproperty if it is an element of this set of sets. Formally, the semantics of HyperLTL formulas is given with respect to a *trace assignment* Π from $\mathcal{V}$ to TR, i.e., a partial function mapping trace variables to actual traces. $\Pi[\pi \mapsto t]$ denotes that π is mapped to t, with everything else mapped according to Π. $\Pi[i,\infty]$ denotes the trace assignment that is equal to $\Pi(\pi)[i,\infty]$ for all π.

$(T,\Pi,i) \vDash a_\pi$	if $a \in \Pi(\pi)[i]$
$(T,\Pi,i) \vDash \neg\varphi$	if $(T,\Pi,i) \nvDash \varphi$
$(T,\Pi,i) \vDash \varphi \vee \psi$	if $(T,\Pi,i) \vDash \varphi$ or $(T,\Pi,i) \vDash \psi$
$(T,\Pi,i) \vDash \bigcirc\varphi$	if $(T,\Pi,i+1) \vDash \varphi$
$(T,\Pi,i) \vDash \varphi\,\mathcal{U}\,\psi$	if $\exists j \geq i.\,(T,\Pi,j) \vDash \psi \wedge \forall i \leq k < j.\,(T,\Pi,k) \vDash \varphi$
$(T,\Pi,i) \vDash \exists\pi.\,\varphi$	if there is some $t \in T$ such that $(T,\Pi[\pi \mapsto t],i) \vDash \varphi$
$(T,\Pi,i) \vDash \forall\pi.\,\varphi$	if for all $t \in T$ it holds that $(T,\Pi[\pi \mapsto t],i) \vDash \varphi$.

3.3 Finite Trace Semantics

We recap the finite trace semantics for HyperLTL [9, 31]. Let $\Pi_{fin} \colon \mathcal{V} \to \Sigma^+$ be a partial function mapping trace variables to finite traces. We define $\epsilon[0]$ as the empty set. By slight abuse of notation, we write $t \in \Pi_{fin}$ to access traces t in the image of Π_{fin}. The satisfaction of a HyperLTL formula φ over a finite trace assignment Π_{fin} and a set of finite traces T, denoted by $(T,\Pi_{fin},i) \vDash \varphi$, is defined as follows:

$(T,\Pi_{fin},i) \vDash a_\pi$	if $a \in \Pi_{fin}(\pi)[i]$
$(T,\Pi_{fin},i) \vDash \neg\varphi$	if $(T,\Pi_{fin},i) \nvDash \varphi$
$(T,\Pi_{fin},i) \vDash \varphi \vee \psi$	if $(T,\Pi_{fin},i) \vDash \varphi$ or $(T,\Pi_{fin},i) \vDash \psi$
$(T,\Pi_{fin},i) \vDash \bigcirc\varphi$	if $\forall t \in \Pi_{fin}.\,\lvert t\rvert > i+1$ and $(T,\Pi_{fin},i+1) \vDash_T \varphi$
$(T,\Pi_{fin},i) \vDash \varphi\,\mathcal{U}\,\psi$	if $\exists j \geq i$ with $j < \min\limits_{t \in \Pi_{fin}} \lvert t\rvert$ such that $(T,\Pi_{fin},j) \vDash \psi$ $\wedge\ \forall k \geq i$ with $k < j$ it holds that $(T,\Pi_{fin},k) \vDash \varphi$
$(T,\Pi_{fin},i) \vDash \exists\pi.\,\varphi$	if there is some $t \in T$ such that $(T,\Pi_{fin}[\pi \mapsto t],i) \vDash \varphi$
$(T,\Pi_{fin},i) \vDash \forall\pi.\,\varphi$	if for all $t \in T$ such that $(T,\Pi_{fin}[\pi \mapsto t],i) \vDash \varphi$

3.4 Monitorability of HyperLTL Specifications

We recap the monitorability definitions for trace properties [35] and hyperproperties [1, 25]. Let $L \subseteq \Sigma^\omega$. We distinguish *good* and *bad* prefixes: $good(L) \coloneqq \{u \in \Sigma^* \mid \forall v \in \Sigma^\omega.\, uv \in L\}$ and $bad(L) \coloneqq \{u \in \Sigma^* \mid \forall v \in \Sigma^\omega.\, uv \notin L\}$, respectively. A trace language L is *monitorable* if every prefix has a (finite) continuation that is either good or bad, formally, $\forall u \in \Sigma^*.\, \exists v \in \Sigma^*.\, uv \in good(L) \vee uv \in bad(L)$.

Theorem 1 ([4]). *Deciding whether an LTL formula φ is monitorable is* PSPACE-*complete.*

Let $H \subseteq \mathcal{P}(\Sigma^\omega)$ be a hyperproperty. We say that a finite set of prefix traces is *good* if every continuation, i.e., a (possibly infinite) set of infinite traces, is contained in H. The set of *good* and *bad prefix traces* is then formally defined as $\mathit{good}(H) \coloneqq \{U \in \mathcal{P}^*(\Sigma^*) \mid \forall V \in \mathcal{P}(\Sigma^\omega).\, U \preceq V \Rightarrow V \in H\}$ and $\mathit{bad}(H) \coloneqq \{U \in \mathcal{P}^*(\Sigma^*) \mid \forall V \in \mathcal{P}(\Sigma^\omega).\, U \preceq V \Rightarrow V \notin H\}$.

Unbounded Sequential Model. A hyperproperty H is *monitorable* in the unbounded input model if every finite prefix set has a good or bad continuation, formally,

$$\forall U \in \mathcal{P}^*(\Sigma^*).\, \exists V \in \mathcal{P}^*(\Sigma^*).\, U \preceq V \wedge \big(V \in \mathit{good}(H) \vee V \in \mathit{bad}(H)\big) \enspace .$$

Theorem 2 ([25]). *Given an alternation-free HyperLTL formula φ. Deciding whether φ is monitorable in the unbounded sequential model is* PSpace*-complete.*

Theorem 3 ([25]). *Deciding whether a HyperLTL formula φ is monitorable in the unbounded sequential model is undecidable.*

Bounded Sequential Model. We give the adapted definition of monitorability and a characterization for alternation-free HyperLTL. A hyperproperty H is *monitorable* in the bounded input model for some bound $b > 0$ if

$$\forall U \in \mathcal{P}^{\leq b}(\Sigma^*).\, \exists V \in \mathcal{P}^b(\Sigma^*).\, U \preceq V \wedge (V \in \mathit{good}^b(H) \vee V \in \mathit{bad}^b(H)) \enspace ,$$

where $\mathit{good}^b(H) \coloneqq \{U \in \mathcal{P}^b(\Sigma^*) \mid \forall V \in \mathcal{P}^b(\Sigma^\omega).\, U \preceq V \Rightarrow V \in H\}$ and $\mathit{bad}(H) \coloneqq \{U \in \mathcal{P}^b(\Sigma^*) \mid \forall V \in \mathcal{P}^b(\Sigma^\omega).\, U \preceq V \Rightarrow V \notin H\}$.

Theorem 4 ([25]). *Deciding whether a HyperLTL formula φ is monitorable in the bounded sequential model is undecidable.*

Parallel Model. Lastly, we consider the parallel model, were b traces are given simultaneously. This model is with respect to monitorability a special case of the bounded model. A hyperproperty H is *monitorable* in the fixed size input model if for a given bound b

$$\forall U \in \mathcal{P}^b(\Sigma^*).\, \exists V \in \mathcal{P}^b(\Sigma^*).\, U \preceq V \wedge (V \in \mathit{good}^b(H) \vee V \in \mathit{bad}^b(H)) \enspace .$$

Theorem 5 ([25]). *Deciding whether a HyperLTL formula φ is monitorable in the parallel model is undecidable.*

4 Algorithms for Monitoring Hyperproperties

We classify the current state-of-the art monitoring algorithms for hyperproperties into two approaches: *combinatorial* approaches [1, 23, 25] and *constraint-based* approaches [9, 31].

As *combinatorial* approaches we understand algorithms that construct monitors by explicitly iterating over each (necessary) combination of traces for monitoring them. For example, consider a trace set T of already monitored traces

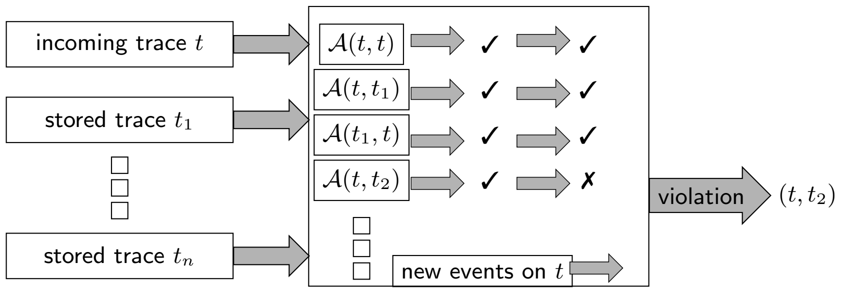

Fig. 5. A combinatorial approach to monitoring hyperproperties [23, 25]: a monitoring template $\mathcal{A}$, constructed from a given HyperLTL formula φ, is initiated with combinations from the new incoming trace t and stored traces $\{t_1, \ldots, t_n\}$. The monitors progress with new events on t, in this case, until a violation is found for trace t and t_2.

and a fresh incoming trace t. A combinatorial monitor would construct each pair $T \times \{t\}$ and check whether the hyperproperty holds on such a trace tuple. The monitor, in the worst case, therefore has to store each incoming trace seen so far. This is currently done by explicit automata constructions, but other methods, such as SAT-solvers could be plugged into such combinatorial approaches as well. In Section 4.1, we will investigate one such approach [25] in detail, which is the algorithmic foundation for the combinatorial algorithm implemented in the current state-of-the-art monitoring tool RVHyper [24].

The *constraint-based* approaches try to avoid the storing of explicit traces by translating the monitoring task into a constraint system. This is currently implemented by rewriting approaches that translate the requirements that a current trace imposes on future traces into the formula. For example, a hyperproperty φ under consideration and a new event e_t on a trace t will be translated into $\varphi[e_t]$ and used as the new specification when monitoring new events on possibly new traces. Such a rewritten formula can then, together with the trace under consideration, be translated into an constraint system, which is fed, for example, into a SAT-solver. In Section 4.2, we will investigate a recently introduced [31] constraint-based algorithm for $\forall^2$ HyperLTL formulas in detail.

4.1 Combinatorial Approaches

Intuition. We describe the automaton-based combinatorial approach introduced in [23, 25] in detail. The basic architecture of the algorithm is depicted in Fig. 5. Let a trace set $T := \{t_1, \ldots, t_n\}$ of already seen traces and a fresh trace t, which is processed online, be given. From a $\forall^*$ HyperLTL formula, a monitor *template* $\mathcal{A}$ is automatically constructed, which runs over two execution traces. This template is then initialized with every combination between t and T. A

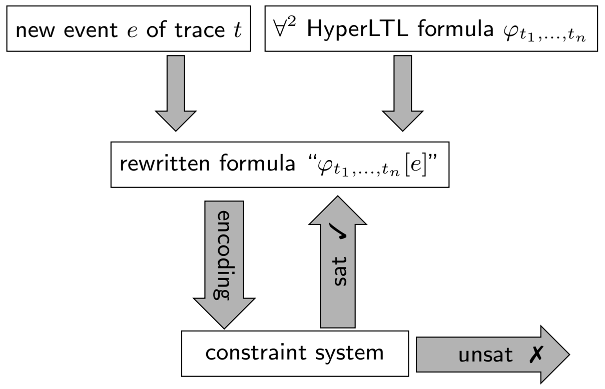
new event e of trace t
$\forall^2$ HyperLTL formula $\varphi_{t_1,\dots,t_n}$
rewritten formula "$\varphi_{t_1,\dots,t_n}[e]$"
encoding
sat ✓
constraint system
unsat ✗

Fig. 6. A constraint-based approach to monitoring hyperproperties [31]: a fresh trace t, and a HyperLTL formula $\varphi_{t_1,\dots,t_n}$, which has already been rewritten with respect to seen traces $t_1,\dots t_n$, will be rewritten to a formula representing the requirements that are posed on future traces. The rewritten formula will be translated into a constraint system, which is satisfiable if the new event complies with the formula $\varphi_{t_1,\dots,t_n}$ and unsatisfiable if there is a violation.

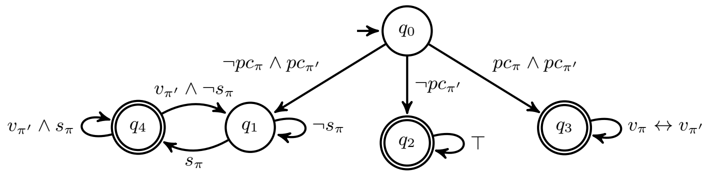
q_0
$\neg pc_\pi \wedge pc_{\pi'}$
$\neg pc_{\pi'}$
$pc_\pi \wedge pc_{\pi'}$
$v_{\pi'} \wedge \neg s_\pi$
$v_{\pi'} \wedge s_\pi$
q_4
q_1
$\neg s_\pi$
s_π
q_2
$\top$
q_3
$v_\pi \leftrightarrow v_{\pi'}$

Fig. 7. [23, 25] Visualization of the monitor template for Formula 3.

violation will be reported when one of the automaton instantiations ends up in a rejecting state.

Example 1 (Conference Management System [23, 25]). Consider a conference management system, where we distinguish two types of traces, *author traces* and *program committee member traces*. The latter starts with proposition pc. Based on these traces, we want to verify that no paper submission is lost, i.e., that every submission (proposition s) is visible (proposition v) to every program committee member in the following step. When comparing two PC traces, we require that they agree on proposition v. The monitor template for the following HyperLTL formalization is depicted in Fig. 7.

$$\forall \pi.\forall \pi'.\, \big((\neg pc_\pi \wedge pc_{\pi'}) \rightarrow \bigcirc\Box(s_\pi \rightarrow \bigcirc v_{\pi'})\big) \wedge \big((pc_\pi \wedge pc_{\pi'}) \rightarrow \bigcirc\Box(v_\pi \leftrightarrow v_{\pi'})\big) \quad (3)$$

Algorithm. Formally, a deterministic monitor template $\mathcal{M} = (\Sigma, Q, \delta, q_0, F)$ [23, 25] is a tuple of a finite alphabet $\Sigma = \mathcal{P}(\mathrm{AP} \times \mathcal{V})$, a non-empty set of states Q, a partial transition function $\delta : Q \times \Sigma \hookrightarrow Q$, a designated initial state

```
input  : $\forall^n$ HyperLTL formula $\varphi$
output: satisfied or $n$-ary tuple witnessing violation
$\mathcal{M}_\varphi = (\Sigma_\mathcal{V}, Q, q_0, \delta, F) =$ build_template($\varphi$);
$T \leftarrow \emptyset$;
$S : T^n \to Q$ initially empty;
while there is a new trace do
    $t \leftarrow \epsilon$;
    for $\boldsymbol{t} \in ((T \cup \{t\})^n \setminus T^n)$ do init $S$ for every new tuple $\boldsymbol{t}$
        $S(\boldsymbol{t}) = q_0$;
    end
    while $p \in \Sigma$ is a new input event do
        $t \leftarrow t\ p$    append $p$ to $t$;
        for $((t_1, \ldots, t_n), q) \in S$ where $t \in (t_1, \ldots, t_n)$ do progress every state in
          $S$
            if $\exists t' \in \{t_1, \ldots, t_n\}. |t'| < |t|$ then some trace ended
                if  $S((t_1, \ldots, t_n)) \in F$ then
                    remove $(t_1, \ldots, t_n)$ from $S$ and continue;
                else
                    return violation and witnessing tuple $\boldsymbol{t}$;
                end
            else if $\delta(S((t_1, \ldots, t_n)), \bigcup_{i=1}^n \bigcup_{a \in t_i[|t|-1]} \{(a, \pi_i)\}) = q'$ then
                $S(N) \leftarrow q'$;
            else
                return violation and witnessing tuple $\boldsymbol{t}$;
            end
        end
    end
    $T = T \cup \{t\}$;
end
return satisfied;
```

Fig. 8. [25] Evaluation algorithm for monitoring $\forall^n$ HyperLTL formulas in the unbounded sequential model.

$q_0 \in Q$, and a set of accepting states $F \subseteq Q$. The instantiated automaton runs in parallel over traces in $\mathcal{P}(\mathrm{AP})^*$, thus we define a run with respect to a n-ary tuple $N \in (\mathcal{P}(\mathrm{AP})^*)^n$ of finite traces. A run of N is a sequence of states $q_0 q_1 \cdots q_m \in Q^*$, where m is the length of the smallest trace in N, starting in the initial state q_0 such that for all i with $0 \leq i < m$ it holds that

$$\delta\left(q_i, \bigcup_{j=1}^{n} \bigcup_{a \in N(j)(i)} \{(a, \pi_j)\}\right) = q_{i+1} \quad .$$

A tuple N is accepted, if there is a run on $\mathcal{M}$ that ends in an accepting state.

The algorithm for monitoring $\forall^n$ HyperLTL formulas in the unbounded sequential model is given in Figure 8. The algorithm proceeds as follows. A

```
input  : HyperLTL formula $Q^n.\psi$
         trace set $T \subseteq \mathcal{P}^*(\Sigma^*)$
output: satisfied or violation
$\mathcal{A}_\psi = (\Sigma_\mathcal{V}, Q, q_0, \delta, F) =$ build_alternating_automaton($\psi$);
if $\bigdiamond_{t_1 \in T} \cdots \bigdiamond_{t_n \in T}$ . LTL_backwards_algorithm($\mathcal{A}_\psi$,$(t_1, t_2, \ldots, t_n)$) then
    return satisfied;
else
    return violation;
end
```

Fig. 9. [25] Offline backwards algorithm for the parallel model, where $\diamondsuit_i := \wedge$ if the i-th quantifier in φ is a universal quantifier and $\vee$ otherwise.

monitoring template is constructed a-priori from the specification (in doubly-exponential time in the size of the formula [14, 38]) and the trace set T is initially empty. For each new trace, we proceed with the incoming events on this trace. The automaton template will then be initialized by each combination between t and traces in T, i.e. $S(\boldsymbol{t}) = q_0$. Each initialized monitor progresses with new input events p until a violation is found, in which case the witnessing tuple $\boldsymbol{t}$ is returned, or a trace ends, in which case this monitor is discarded if no violation occurred. If no violation occurred, and all trace combinations have been monitored, the current trace t is added to the traces that have been seen already, i.e., T.

While the online monitoring algorithms in the bounded sequential and parallel input model can be seen as special cases of the above described algorithm, traces can be processed efficiently in a backwards fashion when considering *offline* monitoring. The algorithm depicted in Fig. 9 exploits the backwards algorithm based on alternating automata [28].

4.2 Constraint-based Approaches

Intuition. We describe the constraint-based monitoring algorithm for $\forall^2$ HyperLTL formulas introduced in [31] in detail. The basic architecture of the algorithm is depicted in Fig. 6. The basic idea is that a formula and an event on a trace will be rewritten into a new formula, which represents the requirements posed on future traces.

Example 2 (Observational Determinism [31]). Assume the event $\{in, out\}$ while monitoring observational determinism: $((out_\pi \leftrightarrow out_{\pi'})\,\mathcal{W}(in_\pi \not\leftrightarrow in_{\pi'}))$. The formula is rewritten by applying the standard expansion laws and inserting $\{in, out\}$ for the atomic propositions indexed by the trace variable π: $\neg in \vee out \wedge \bigcirc((out_\pi \leftrightarrow out_{\pi'})\,\mathcal{W}(in_\pi \not\leftrightarrow in_{\pi'}))$. Based on this, a Boolean constraint system is built incrementally: one starts by encoding the constraints corresponding to the LTL part $\neg in \vee out$ and encodes the HyperLTL part as

```
Input  : $\forall\pi,\pi'.\,\varphi$, $T \subseteq \Sigma^+$
Output: violation or no violation
$\psi \coloneqq \mathtt{nnf}(\hat{\varphi})$
$C \coloneqq \top$
foreach $t \in T$ do
    $C_t \coloneqq v_{\psi,0}$
    $t_{enc} \coloneqq \top$
    while $e_i \coloneqq \mathit{getNextEvent}(t)$ do
        $t_{enc} \coloneqq t_{enc} \wedge \mathtt{encoding}(e_i)$
        foreach $v_{\phi,i} \in C_t$ do
            $c \coloneqq \psi[\pi, e_i, i]$
            $C_t \coloneqq C_t \wedge (v_{\phi,i} \rightarrow\ c)$
        end
        if $\neg \mathit{sat}(C \wedge C_t \wedge t_{enc})$ then
            return violation
        end
    end
    foreach $v^+_{\phi,i+1} \in C_t$ do
        $C_t \coloneqq C_t \wedge v^+_{\phi,i+1}$
    end
    foreach $v^-_{\phi,i+1} \in C_t$ do
        $C_t \coloneqq C_t \wedge \neg v^-_{\phi,i+1}$
    end
    $C \coloneqq C \wedge C_t$
end
return no violation
```

Fig. 10. [31] Constraint-based algorithm for monitoring $\forall^2$HyperLTL formulas.

variables. Those variables will then be defined incrementally when more elements of the trace become available. A violation will be reported when the constraint system becomes unsatisfiable.

Algorithm. We define the operation $\varphi[\pi, e, i]$ (taken from [31]), where $e \in \Sigma$ is an event and i is the current position in the trace, as follows: $\varphi[\pi, e, i]$ transforms φ into a propositional formula, where the variables are either indexed atomic propositions p_i for $p \in AP$, or a variable $v^-_{\varphi',i+1}$ and $v^+_{\varphi',i+1}$ that act as placeholders until new information about the trace comes in. Whenever the next event e' occurs, the variables are defined with the result of $\varphi'[\pi, e', i+1]$. If the trace ends, the variables are set to *true* and *false* for v^+ and v^-, respectively. In Fig. 11, we define $\varphi[\pi, e, i]$ of a $\forall^2$HyperLTL formula $\forall\pi,\pi'.\,\varphi$ in NNF, event $e \in \Sigma$, and $i \geq 0$ recursively on the structure of the body φ. We write $v_{\varphi,i}$ to denote either $v^-_{\varphi,i}$ or $v^+_{\varphi,i}$.

The algorithm for monitoring $\forall^2$ HyperLTL formulas with the constraint-based approach is given in Fig. 10. We continue with the explanation of the algorithm (taken from [31]): ψ is the negation normal form of the symmetric

$$
\begin{array}{llll}
a_\pi[\pi,e,i] & := \begin{cases} \top & \text{if } a \in e \\ \bot & \text{otherwise} \end{cases} & (\neg a_\pi)[\pi,e,i] & := \begin{cases} \top & \text{if } a \notin e \\ \bot & \text{otherwise} \end{cases} \\
a_{\pi'}[\pi,e,i] & := a_i & (\neg a_{\pi'})[\pi,e,i] & := \neg a_i \\
(\varphi \vee \psi)[\pi,e,i] & := \varphi[\pi,e,i] \vee \psi[\pi,e,i] & (\varphi \wedge \psi)[\pi,e,i] & := \varphi[\pi,e,i] \wedge \psi[\pi,e,i] \\
(\bigcirc \varphi)[\pi,e,i] & := v^-_{\varphi,i+1} & (\bigcirc_{\mathrm{w}} \varphi)[\pi,e,i] & := v^+_{\varphi,i+1}
\end{array}
$$

$$
\begin{array}{ll}
(\varphi \,\mathcal{U}\, \psi)[\pi,e,i] & := \psi[\pi,e,i] \vee (\varphi[\pi,e,i] \wedge v^-_{\varphi \mathcal{U} \psi,i+1}) \\
(\varphi \,\mathcal{R}\, \psi)[\pi,e,i] & := \psi[\pi,e,i] \wedge (\varphi[\pi,e,i] \vee v^+_{\varphi \mathcal{R} \psi,i+1})
\end{array}
$$

Fig. 11. [31] Recursive definition of the rewrite operation.

closure of the original formula. We build two constraint systems: C containing constraints of previous traces and C_t (built incrementally) containing the constraints for the current trace t. Consequently, we initialize C with $\top$ and C_t with $v_{\psi,0}$. If the trace ends, we define the remaining v variables according to their polarities and add C_t to C. For each new event e_i in the trace t, and each "open" constraint in C_t corresponding to step i, i.e., $v_{\phi,i} \in C_t$, we rewrite the formula ϕ and define $v_{\phi,i}$ with the rewriting result, which, potentially introduced new open constraints $v_{\phi',i+1}$ for the next step $i+1$. The constraint encoding of the current trace is aggregated in constraint t_{enc}. If the constraint system given the encoding of the current trace turns out to be unsatisfiable, a violation to the specification is detected, which is then returned.

5 Optimizations

Both monitoring approaches rely heavily on optimization techniques to become feasible in practice. Naive implementations, that blindly store all traces seen so far or consider the same constraints multiple times, will run out of memory quickly or will take unfeasibly long. We present several techniques that significantly speed up the monitoring process.

5.1 Specification Analysis

We can analyze the specification and determine if it is symmetric, transitive, or reflexive. Formally, we define symmetry of a HyperLTL formulas as follows (reflexivity and transitivity is discussed in detail in [25]).

Definition 1 ([25]). *Let ψ be the quantifier-free part of some HyperLTL formula φ over trace variables $\mathcal{V}$. We say φ is invariant under trace variable permutation $\sigma : \mathcal{V} \to \mathcal{V}$, if for any set of traces $T \subseteq \Sigma^\omega$ and any assignment $\Pi : \mathcal{V} \to T$, $(\emptyset, \Pi, 0) \vDash \psi \Leftrightarrow (\emptyset, \Pi \circ \sigma, 0) \vDash \psi$. We say φ is symmetric, if it is invariant under every trace variable permutation in $\mathcal{V}$.*

Observational determinism, for example, is symmetric. To illustrate the impact of this observation, consider again Fig. 5. Symmetry means that one of the automaton instantiation $\mathcal{A}[t, t_i]$ or $\mathcal{A}[t_i, t]$ can be omitted for each $i \leq n$, resulting in an reduction of half the monitor instantiations.

A HyperLTL formula can be checked for symmetry, transitivity and reflexivity fully automatically and a-priori to the monitoring task with a satisfiability solver for hyperproperties, such as EAHyper [22]. Such a check, for example for observational determinism, is performed in under a second.

5.2 Trace Analysis

Keeping the set of stored traces minimal is crucial for a combinatorial approach to monitoring hyperproperties: We explain a method that checks whether a trace t poses strictly stronger requirements on future traces than another trace t'. In this case, t' could be safely discarded without losing the ability to detect every violation of the hyperproperty.

Definition 2 ([25]). *Given a HyperLTL formula φ, a trace set T and an arbitrary $t \in \Sigma^\omega$, we say that t is (T, φ)-redundant if T is a model of φ if and only if $T \cup \{t\}$ is a model of φ as well, formally*

$$\forall T' \supseteq T.\, T' \in \mathcal{H}(\varphi) \Leftrightarrow T' \cup \{t\} \in \mathcal{H}(\varphi) \enspace .$$

Example 3 ([31]). Consider the monitoring of the HyperLTL formula $\forall \pi, \pi'. \Box(a_\pi \rightarrow \neg b_{\pi'})$, which states that globally if a occurs on any trace π, then b is not allowed to hold on any trace π', on the following incoming traces:

$$\boxed{\{a\}}\boxed{\{\}}\boxed{\{\}}\boxed{\{\}} \qquad \neg b \textit{ is enforced on the 1st pos.} \tag{4}$$

$$\boxed{\{a\}}\boxed{\{a\}}\boxed{\{\}}\boxed{\{\}} \qquad \neg b \textit{ is enforced on the 1st and 2nd pos.} \tag{5}$$

$$\boxed{\{a\}}\boxed{\{\}}\boxed{\{a\}}\boxed{\{\}} \qquad \neg b \textit{ is enforced on the 1st and 3rd pos.} \tag{6}$$

In this example, the requirements of the first trace are dominated by the requirements of the second trace, namely that b is not allowed to hold on the first and second position of new incoming traces. Hence, the first trace must not be stored any longer to detect a violation.

5.3 Tree Maintaining Formulas and Conjunct Splitting

For constraint-based approaches, a valuable optimization is to store formulas and their corresponding variables in a tree structure, such that a node corresponds to an already seen rewrite. If a rewrite is already present in the tree, there is no need to create any new constraints. By splitting conjuncts in HyperLTL formulas, we can avoid introducing unnecessary nodes in the tree.

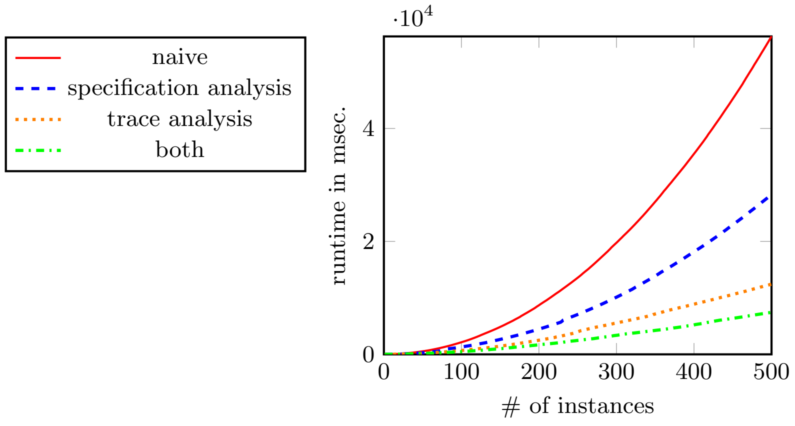

Fig. 12. [23, 25] Hamming-distance preserving encoder: runtime comparison of the naive monitoring approach with different optimizations and the combination thereof.

Table 1. [31] Average results of BDD and SAT based constraint-based algorithms compared to the combinatorial algorithm on traces generated from circuit instances. Every instance was run 10 times.

instance	# traces	length	time combinatorial	time SAT	time BDD
XOR1	19	5	12ms	47ms	49ms
XOR2	1000	5	16913ms	996ms	1666ms
counter1	961	20	9610ms	8274ms	303ms
counter2	1353	20	19041ms	13772ms	437ms
MUX1	1000	5	14924ms	693ms	647ms
MUX2	80	5	121ms	79ms	81ms

Example 4 ([31]). Consider $\forall \pi, \pi'. \varphi$ with $\varphi = \square((a_\pi \leftrightarrow a'_\pi) \lor (b_\pi \leftrightarrow b'_\pi))$, which demands that on all executions on each position at least on of propositions a or b agree in its evaluation. Consider the two traces $t_1 = \{a\}\{a\}\{a\}$, $t_2 = \{a\}\{a, b\}\{a\}$ that satisfy the specification. As both traces feature the same first event, they also share the same rewrite result for the first position. Interestingly, on the second position, we get $(a \lor \neg b) \land s_\varphi$ for t_1 and $(a \lor b) \land s_\varphi$ for t_2 as the rewrite results. While these constraints are no longer equal, by the nature of invariants, both feature the same subterm on the right hand side of the conjunction. We split the resulting constraint on its syntactic structure, such that we would no longer have to introduce a branch in the tree.

6 Experimental Results

The presented algorithms and optimizations implemented in RVHyper [24] were extensively evaluated over the last years [23, 24, 31, 25].

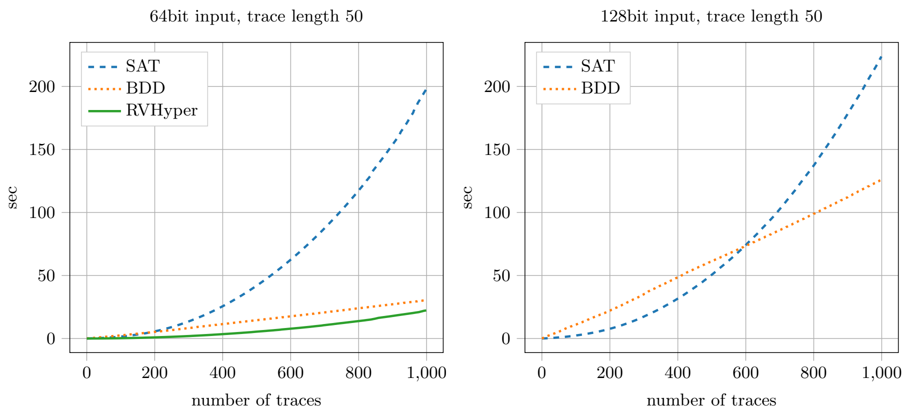

Fig. 13. [31] Runtime comparison between the combinatorial algorithm and the constraint-based algorithm implemented in RVHyper on a non-interference specification with traces of varying input size.

A first benchmark that shows the impact of the trace and specification analysis is the following: it is monitored whether an encoder preserves a Hamming-distance of 2 [25], which can be encoded as a universally quantified HyperLTL formula [11]: $\forall \pi \pi'. (\diamondsuit (I_\pi \not\leftrightarrow I_{\pi'}) \rightarrow ((O_\pi \leftrightarrow O_{\pi'}) \mathcal{U} ((O_\pi \not\leftrightarrow O_{\pi'}) \wedge \bigcirc ((O_\pi \leftrightarrow O_{\pi'}) \mathcal{U} (O_\pi \not\leftrightarrow O_{\pi'})))))$. In Fig. 12 a comparison between the naive monitoring approach and the monitor using specification analysis and trace analysis, as well as a combination thereof is depicted. Traces were built randomly, where the corresponding bit on each position had a 1% chance of being flipped.

A second benchmark was introduced in [24] with the idea to detect spurious dependencies in hardware design. Traces were generated from circuit instances and then monitored whether input variables influence out variables. The property was specified as the following HyperLTL formula: $\forall \pi_1 \forall \pi_2. (\boldsymbol{o}_{\pi_1} \leftrightarrow \boldsymbol{o}_{\pi_2}) \mathcal{W} (\bar{\boldsymbol{i}}_{\pi_1} \not\leftrightarrow \bar{\boldsymbol{i}}_{\pi_2})$, where $\bar{\boldsymbol{i}}$ denotes all inputs except $\boldsymbol{i}$. The results are depicted in Table 1.

The next benchmark [31] considers non-interference [33], which is an important information flow policy demanding that an observer of a system cannot infer any high security input of a system by observing only low security input and output. Reformulated we could also say that all low security outputs $\boldsymbol{o}^{low}$ have to be equal on all system executions as long as the low security inputs $\boldsymbol{i}^{low}$ of those executions are the same: $\forall \pi, \pi'. (\boldsymbol{o}^{low}_{\pi} \leftrightarrow \boldsymbol{o}^{low}_{\pi'}) \mathcal{W} (\boldsymbol{i}^{low}_{\pi} \not\leftrightarrow \boldsymbol{i}^{low}_{\pi'})$. The results of the experiments are depicted in Fig. 13. For 64 bit inputs, the BDD implementation performs well when compared to the combinatorial approach, which statically constructs a monitor automaton. For 128 bit inputs, it was not possible to construct the automaton for the combinatorial approach in reasonable time.

The last benchmark considers *guarded invariants*, which express a certain invariant relation between two traces, which are, additionally, guarded by a precondition. Fig. 14 shows the results of monitoring an arbitrary invariant P :

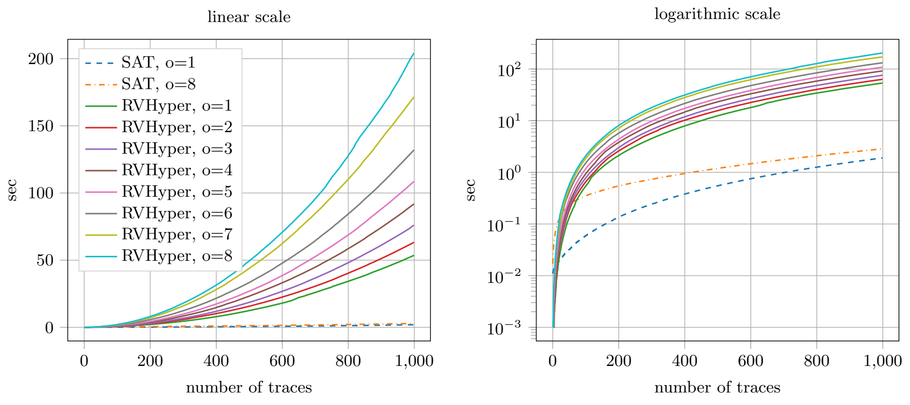

Fig. 14. [31] Runtime comparison between the combinatorial approach and the constraint-based monitor on the guarded invariant benchmark with trace lengths 20, 20 bit input size.

$\Sigma \to \mathbb{B}$ of the following form: $\forall \pi, \pi'. \diamondsuit(\vee_{i \in I} i_\pi \not\leftrightarrow i_{\pi'}) \to \square(P(\pi) \leftrightarrow P(\pi'))$. The constraint-based approach significantly outperforms combinatorial approaches on this benchmark as the conjunct splitting optimization synergizes well with current SAT-solver implementations.

7 Conclusion

We classified current monitoring approaches into *combinatorial* and *constraint-based* algorithms and explained their basic architecture. We have gone into detail into two of these approaches and summarized current optimization technique making the monitoring of hyperproperties feasible in practice.

Future work consists of implementing and adapting more optimization techniques for constraint-based and combinatorial approaches. It would also be interesting to plug SAT and SMT solvers into combinatorial monitoring approaches, instead of using automata. Furthermore, considering the monitoring problem of specifications given in HyperQPTL, i.e., the extension of HyperLTL with quantification over propositions, is not studied yet. This problem is particularly interesting and challenging since HyperQPTL allows for a true combination of ω-regular properties and hyperproperties.

Acknowledgements

This paper is based on a tutorial that will be given at the 19th International Conference on Runtime Verification. The work summarized here has previously appeared in various publications [23, 24, 31, 25]. The author is particularly grateful to his coauthors Bernd Finkbeiner, Marvin Stenger, and Leander Tentrup and, furthermore, to Maximilian Schwenger for his valuable comments on an earlier version of this paper.

References

1. Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proceedings of CSF*. IEEE Computer Society, 2016.
2. Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of CSF*. IEEE Computer Society, 2009.
3. Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of PLAS*. ACM, 2010.
4. Andreas Bauer. Monitorability of omega-regular languages. *CoRR*, 2010.
5. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in WebKit's javascript bytecode. In *Proceedings of POST*, LNCS. Springer, 2014.
6. Borzoo Bonakdarpour and Bernd Finkbeiner. Runtime verification for hyperltl. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, 2016.
7. Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *Proceedings of CSF*. IEEE Computer Society, 2018.
8. Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proceedings of II*, LNCS. Springer, 2018.
9. Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proceedings of TACAS*, LNCS, 2017.
10. Andrey Chudnov, George Kuan, and David A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *Proceedings of CSF*. IEEE Computer Society, 2014.
11. Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Proceedings of POST*, LNCS. Springer, 2014.
12. Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, (6), 2010.
13. Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. The hierarchy of hyperlogics. In *To appear in the proceedings of LICS*, 2019.
14. Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In *Proceedings of CAV*, LNCS. Springer, 2005.
15. Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of SP*. IEEE Computer Society, 2010.
16. Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of VMCAI*, LNCS. Springer, 2012.
17. Rayna Dimitrova, Bernd Finkbeiner, and Markus N. Rabe. Monitoring temporal information flow. In *Proceedings of ISoLA*, LNCS. Springer, 2012.
18. E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 1986.
19. Bernd Finkbeiner and Christopher Hahn. Deciding hyperproperties. In *Proceedings of CONCUR*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
20. Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. Mghyper: Checking satisfiability of hyperltl formulas beyond the exists forall fragment. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA*, 2018.

21. Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. Synthesizing reactive systems from hyperproperties. In *Computer Aided Verification - 30th International Conference, CAV*, 2018.
22. Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. EAHyper: satisfiability, implication, and equivalence checking of hyperproperties. In *Proceedings of CAV*, LNCS. Springer, 2017.
23. Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. In *Proceedings of RV*, LNCS. Springer, 2017.
24. Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Rvhyper: A runtime verification tool for temporal hyperproperties. In *Proceedings of TACAS*, LNCS. Springer, 2018.
25. Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, 2019.
26. Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In *Computer Aided Verification - 30th International Conference, CAV 2018*, 2018.
27. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of CAV*, LNCS. Springer, 2015.
28. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2), 2004.
29. Bernd Finkbeiner and Martin Zimmermann. The first-order logic of hyperproperties. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, 2017.
30. Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of ASIAN*, LNCS. Springer, 2006.
31. Christopher Hahn, Marvin Stenger, and Leander Tentrup. Constraint-based monitoring of hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS*, 2019.
32. Máté Kovács and Helmut Seidl. Runtime enforcement of information flow security in tree manipulating processes. In *Proceedings of ESSoS*, LNCS. Springer, 2012.
33. John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, (1), 1992.
34. Amir Pnueli. The temporal logic of programs. In *Proceedings of FOCS*. IEEE Computer Society, 1977.
35. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proceedings of FM*, LNCS. Springer, 2006.
36. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, (1), 2003.
37. G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*. ACM, 2004.
38. Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, (3), 2012.
39. Thomas. Path logics with synchronization. In *Perspectives in Concurrency Theory*, 2009.
40. Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of CSF*. IEEE Computer Society, 2014.